**CAB301 Assignment 1 – Empirical Analysis of an Algorithm**

**Andrew Grant | N9167340**

**Submitted: 15th April 2017**

**Summary**

This report analyses the average case efficiency of the provided ‘brute force median’ algorithm as seen in figure 1. It does this through experiments run on the occurrence of the Algorithm’s basic operation and increase in execution time as the size of the test array increases. The algorithm has been implemented in the form of function inside a C++ program. Graphs and code snippets are provided as *figures* below the references section and they are referred to throughout this report. The end test results show a very high correlation with the estimated theoretical efficiency for this algorithm.

**1. Description of the Algorithm**

The ‘brute force median’ algorithm (figure 1), as the name implies, determines which element in a given array is the median by using a ‘brute force’ search and compare method. If the array size/length is odd, then it will accurately return the element that is the middle value (median). However if the length of the array is even then instead of averaging the two middle values, it will simply select the first or ‘left most’ of the two middle elements. This means that it will not accurately provide the correct median in the case of an Array with an even length.

When called, the Algorithm takes in an Array (*A*) as a parameter. It then identifies a ceiling midpoint for the set of values in *A* and assigns that midpoint value to *k*. This midpoint is later used as a comparative reference to determine if the selected element of the array (*A[i])* is indeed the median. Each element in *A* is then sequentially compared against Each other element. Counters are initialised and are incremented when the selected element (*A[i])* is either greater than or equal to the element it is being compare against. After *A[i]* has been compared to all other elements in the array (including itself) the counters are then compared against the midpoint value *k*. If the number of elements smaller *A[i]* is less than *k,* and if the sum of the elements that are either smaller than or equal in value to *A[i]* is not greater than *k,* then *A[i]* is the median (At least in the case of odd numbers as examined above) and it’s value is returned. If those conditions are not met, the next element in the array is selected, (becoming the new *A[i]*) and the process continues until the median can be determined

**2. Identifying the Algorithm’s Basic Operation**

The basic operation in this algorithm is highlighted in figure 1; it is the comparison of the *‘jth’* element to the *‘ith’* element in the array, inside the inner for-loop, to determine if the value of *j* is less than the value of *i*. It is the operation that is completed for every element in the array multiple times until the *‘ith’* element has been identified as the Median. This comparison loop will be performed for every element in the array, irrespective of what position the median is located, until the median is found. This tells us that even in the best-case scenario, this comparison operation will be performed n times (n being the size/length of the array that is being fed into the algorithm). This is the most strenuous part of the algorithm and is the operation would have the greatest influence on the algorithm’s execution time.

The reason the basic operation is not the other comparison of *j* to *i* (to determine whether their values are equal) is because this operation is not performed if the first comparison is successful. Thus it will have less impact on the overall execution time than the initial comparison operation.

**3. Determining the Average-Case Efficiency**

The estimated average-case efficiency of this ‘brute force median’ algorithm is **Cavg(n) = n(n+1)/2.** This is based on the average-case efficiency for a sequential search which is identified as *‘Cavg(n) = (n+1)/2.’* [1, pp. 47-49]. This is because once the inner loop of the algorithm is hit, it essentially becomes a sequential search. The only difference here is that the basic operation is performed for each element of the array, regardless of what position the median is located at, before determining if the selected element is the median. That is why our efficiency function is multiplied by n; to account for the extra searches that are being performed for each loop.

From this we can also identify that the algorithm’s efficiency class and order of growth is quadratic [Θ(n²)] due to its worst case is the basic operation being performed *n²* times. This also gives us an idea of what type of trend will appear when our test results are plotted.

**4. Methodology, Tools and Techniques**

All tests were performed on an Asus laptop computer (AMD A6-5200 APU, 4.00GB RAM) running the Windows 10 operating system.

The ‘brute force median’ algorithm was implemented into its own function of a C++ application created in the freely available *Code::Blocks* IDE [figure 4]. In the ‘*main*’ of my C++ application I export the data from my tests into a .csv file using the method described in reference [3], [figures 5 & 6]. That file is the opened in Microsoft Excel where the data can be viewed and graphs generated [figures 2 & 3].

The C++ *random* Library was utilised with help from reference [2] below to generate random numbers to populate my test arrays. To ensure unique values were generated each time, I used the ‘*windows.h’* library’s *GetTickCount()* function as a seed for my RNG*.* Similarly the C++ 11 library, ‘*chrono’,* was used with assistance from reference [4] to provide an accurate measure of the algorithms execution time in milliseconds.

**5. Experimental Results**

In this final section we will refer to the figures found below to discuss the results of the tests and compare them to the estimations made about the algorithm’s average-case efficiency in section 3 above.

Figure 4 shows the implementation of the ‘brute force median’ algorithm [Figure 1] into a function of my C++ program. I ensured it’s functional correctness by manually testing arrays of unique size and element order. I had my original *main()* function output to the screen the returned median of the provided array as well as the total number of basic operations performed in order to determine the median. These initial test results (including upper and lower bound tests) were manually checked by hand for confirmation that the implemented algorithm was performing as intended.

**5.1 Average-Case Basic Operations**

Once I was satisfied with functional correctness of my implementation I began working towards the *main()* function as seen in figure 5. Several loops were set up in order to generate a large number test data with a single run of the *main()* function.

The outermost loop would be used to increase the size/length of the test Array by the set increment. My basic operation tests used an array length of 100 which was looped though to 10,000 in increments of 10. I determined that values of array lengths less than 100 produced insignificant results not worth considering to create an accurate trend, and for the same reason decided to increment the array size by 10 instead of by 1 for each loop. Tests were unnecessary beyond 10,000 as this was sufficient to produce an accurate trend to confirm the estimated efficiency function. The average ‘operations performed’ count was then exported to the .csv file in the last step of this loop.

The middle loop was used to determine how many tests should be run for each array size tested. I settled for 100 arrays for each array size tested and then took the average to most accurately determine the average execution time and number of basic operations performed while keeping the duration of the tests at a reasonable completion time.

The innermost loop was simply used to generate all of the elements for the test array according to the required array size for that iteration of the loop. As explained above, implementation of the RNG code was obtained from reference [3].

The code used for my *main()* function in figure 5, produced the test data used to plot the graph in figure 2. In that graph you can also see (in orange) the plot of the quadratic function [n(n+1)/2] that was estimated to be the efficiency when I analysed the algorithm in section 3. It is clear from this that a trend has been observed that highly matches the estimated efficiency of the algorithm.

**5.2 Average-Case Execution Time.**

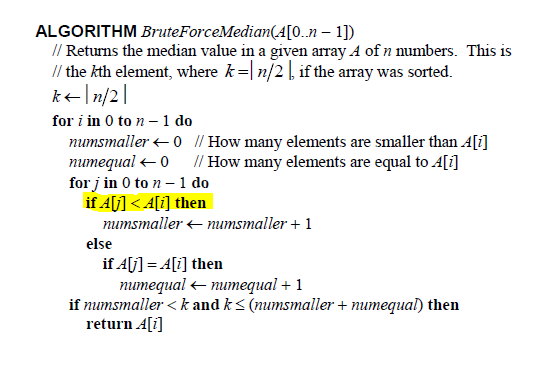
When implementing the tests to measure execution time instead of basic operations performed. I rewrote the *main()* function to how it appears in figure 6. The same explanation as above (when discussing the outermost loop) can be used for my decision to measure average execution times from array sizes of 400 – 10,000 in increments of 50. I started with array sizes of 400, because values smaller than that produced negligible data under 0 milliseconds that would not assist in identifying the estimated trend. Increments of 50 were used in these tests instead of 10 primarily to save time on running the tests but also because the ultimate trend in data produced, when graphed, was sufficient for those larger increments [figure 3].

Nothing was changed in the middle and innermost loops besides from the inclusion of *‘TIMER’* calls which were used to record the execution time, each time the algorithm function was called. These used the C++ 11 ‘*chrono’* library and implementations from reference [4]. See figure 7 to see how these *‘TIMER’* calls are defined. All three are quite self-explanatory. My modified *STOP\_TIMER* also records the execution time by incrementing the *‘etime’* counter used to take the average during the .csv export step of the outermost loop.

The code used for my *main()* function in figure 6, produced the test data used to plot the graph in figure 3. It is quite clear when compared to the graph in figure 2, that the trend being produced is the same quadratic growth as seen in the estimated efficiency function from section 3.

**References**

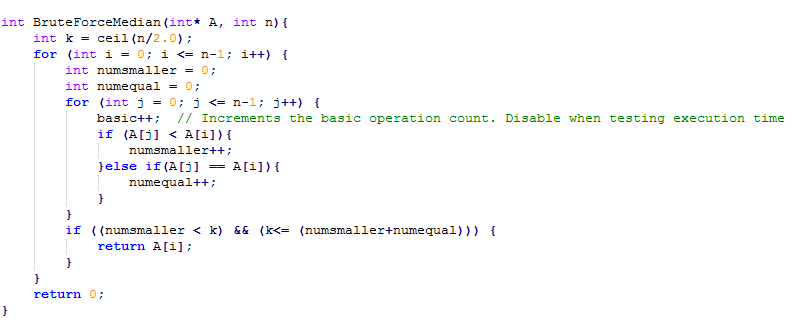
1. A. Levitin. *Introduction to the Design and Analysis of Algorithms.* Addison-Wesley, second edition, 2007. ISBN 0-321-36413-9.
2. *std::uniform\_int\_distribution.* cppreference.com, 2017. http://en.cppreference.com/w/cpp/numeric/random/uniform\_int\_distribution. - **For help with RNG.**
3. A. Allain. *C++ File I/O.* Cprogramming.com, 2011. http://www.cprogramming.com/tutorial/lesson10.html - **For help with writing to .csv file.**
4. E. Smistad. *Measuring runtime in milliseconds using the C++ 11 chrono library.* Erik Smistad, 2012. https://www.eriksmistad.no/measuring-runtime-in-milliseconds-using-the-c-11-chrono-library/ - **For help recording execution time.**

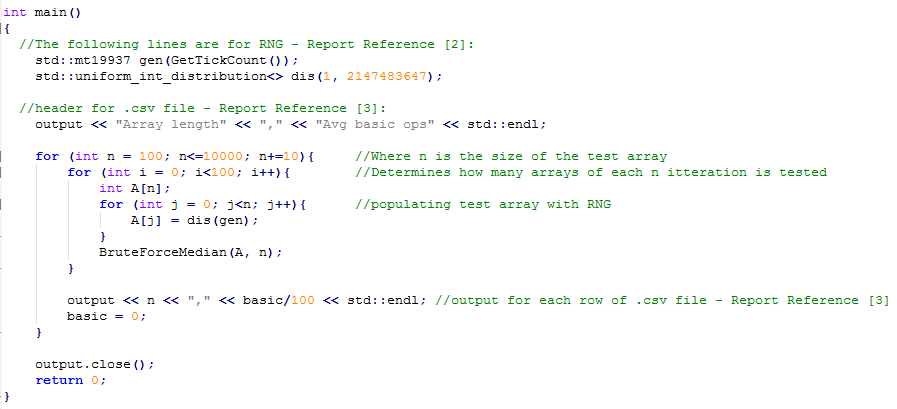


**Figure 1:** The provided ‘brute force median’ algorithm with the identified basic operation highlighted.

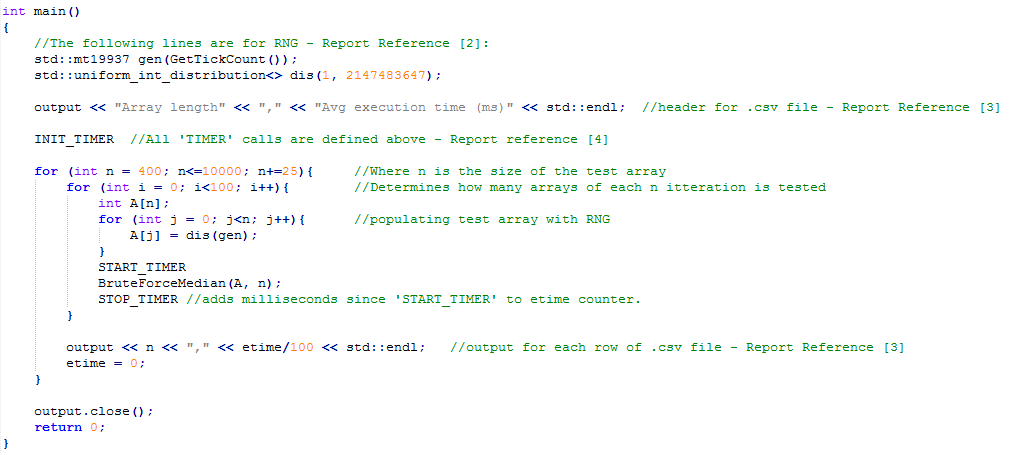
**Figure 2:** Plot of the average operation count from a test array size of 100 – 10,000. The results were obtained by taking the average operations performed for 100 unique arrays for each 10th array size up to 10,000. Also plotted was the average efficiency function [n(n+1)/2] estimated in section 3 of the report.

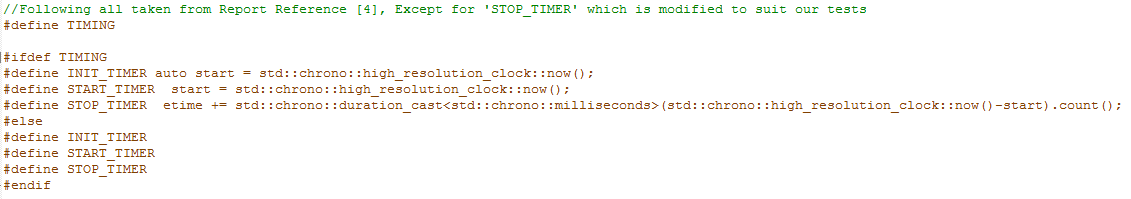
**Figure 3:** Plot of the Average execution time from a test array of size 400 – 10,000. The results were obtained by taking the average operations performed for 100 unique arrays for each 25th array size up to 10,000. Note that although this test was slightly less extensive, the same trend is clearly observed as in figure 2 above.

**Figure 4:** Algorithm implementation in C++ function. Correctly follows the provided algorithm from figure 1 with the exception of an included counter which is used to determine the basic operations performed when the function is called.



**Figure 5:** The *main()* function when test is set up to record basic operations performed instead of average execution time.

**Figure 6:** The *main()* function when test is set up to record average execution time instead of basic operations performed. Notice the addition of ‘TIMER’ calls, they are modified from examples given at Reference [4]. See figure 7 bellow.



**Figure 7:** How the C++11 *chrono* library was used to record average execution time in the program. Implementation was a modified version of that observed in Reference [4] of this report.